Extending a Compiler Backend for Complete Memory Error Detection

Norman A. Rink[1] Jeronimo Castrillon[2]

Abstract: Technological advances drive hardware to ever smaller feature sizes, causing devices to become more vulnerable to faults. Applications can be protected against errors resulting from faults by adding error detection and recovery measures in software. This is popularly achieved by applying automatic program transformations. However, transformations applied to intermediate program representations are fundamentally incapable of protecting against vulnerabilities that are introduced during compilation. In particular, the compiler backend may introduce additional memory accesses. This report presents an extended compiler backend that protects these accesses against faults in the memory system. It is demonstrated that this enables the detection of all single bit flips in memory. On a subset of SPEC CINT2006 the runtime overhead caused by the extended backend amounts to 1.50*x* for the 32-bit processor architecture *i386*, and 1.13*x* for the 64-bit architecture *x86_64*.

Keywords: transient hardware faults, soft errors, memory errors, error detection, fault tolerance, resilience, compiler backend, code generation, intermediate representation (IR), LLVM

1 Introduction

Aggressive technology scaling increases the rates of hardware faults [Sh02, Bo05, Bl06, Ba05], and faults cause erroneous application behavior with non-negligible probabilities [SPW09, NDO11]. Transient hardware faults, also known as *soft errors*, are commonly attributed to cosmic radiation [Ba05]. However, due to shrinking feature sizes, devices are also becoming more vulnerable to variations in supply voltage and temperature, which reduces reliability [Bo05, Sh14]. Moreover, the current trends toward lowering energy consumption and temperature dissipation further reduce reliability [Es11, Ta12, Sh14].

In safety-critical automotive applications, faults that go undetected can pose a danger to human life. Therefore, software that is designed for applications with strict safety and reliability requirements must incorporate measures to tolerate hardware faults [Pa08]. Software can be made fault-tolerant by adding integrity checks to programs. When a check fails, an error has been detected and suitable measures can be taken to recover from it. To enable checks, and hence error detection, some form of redundancy must be added to programs. This can be done conveniently by applying automatic program transformations, such as source-to-source transformations, cf. [Re99, BSS13, KF15, Ka16]. With the rising popularity of the LLVM framework and intermediate representation (IR) [LA04],

[1] Center for Advancing Electronics Dresden, Technische Universität Dresden, Chair for Compiler Construction, 01062 Dresden, norman.rink@tu-dresden.de
[2] Center for Advancing Electronics Dresden, Technische Universität Dresden, Chair for Compiler Construction, 01062 Dresden, jeronimo.castrillon@tu-dresden.de

many fault tolerance schemes have appeared that are implemented as IR transformations, e.g. [FSS09, Sc10, Fe10, Zh10, Ri15, CNV16]. Operating on IR has the advantages of target-independence and increased productivity compared with operating on machine instructions. However, when transformations are applied to programs at an abstraction level above machine instructions, the compiler backend may introduce new vulnerabilities to faults. Specifically, the backend introduces numerous additional memory accesses, cf. Figure 1, which are then not protected against faults in the memory system.

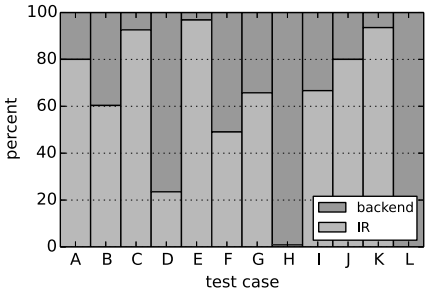


Fig. 1: Dynamic load operations present in the intermediate representation (IR) of programs or inserted by the compiler backend. The test programs labeled A–L are introduced in Section 4, cf. Table 1.

In this work we present a modified compiler backend that adds error detection measures to the memory accesses it inserts. By combining this backend with any fault tolerance scheme that operates on program IR or source code, errors can be detected in all memory accesses that occur in the final machine code. Our compiler backend implements error detection by *dual modular redundancy* (DMR), i.e. data words in memory are duplicated. Whenever a data word is loaded from memory, the duplicated copies are compared. If disagreement is found, an error has been detected. Thus, DMR is capable of detecting any number of flipped bits within a data word. For complete protection of all memory accesses in the final machine code of programs, the extended compiler backend will be combined with the *AN encoding* error detection scheme at the IR level [Br60, Fo89, FSS09, Ri15].

Many fault tolerance schemes circumvent the problem of memory errors by assuming that memory is protected against faults by hardware measures, such as memory modules equipped with *error correcting codes* (ECC) [Re05, YGS09, MPC14, DS16]. This assumption, however, is problematic for two reasons. First, cost and area considerations may rule out using ECC memory at all levels in the memory hierarchy, especially in on-chip caches and load-store queues. In fact, the need to protect a processor's load-store queue against faults has recently been stressed [DS16]. Second, it has been found that the widely used *single error correcting, double error detecting* (SECDED) codes are incapable of handling large fractions of error patterns that occur in practice [HSS12].

This article is structured as follows. Section 2 introduces memory faults and error detection schemes, including the AN encoding scheme. Section 3 identifies the memory accesses that are inserted by the compiler backend and explains how they are equipped with error

detection measures. Section 4 evaluates our error detection scheme. Section 5 discusses related work, and Section 6 summarizes and discusses the findings of the present work.

2 Background

Faults in memory are a major cause of erroneous application behavior and service disruptions [SPW09, HSS12]. Typical faults in memory cells are bit flips caused by energetic particles that originate from cosmic radiation [Ba05]. However, motivated by the current trend toward reducing energy consumption, it has been suggested that the operating voltage of SRAM be lowered [Es12], and that refresh cycles of DRAM modules be extended [Li11, We15]. Both suggestions reduce the capability to retain data and hence increase the probability of memory faults.

The probability that a data word is corrupted by a fault increases with the time that the data word spends in memory. When considering fault tolerance measures for the memory system, main memory is targeted first since this is where the lifetimes of data will generally be the longest. This also means that when, say, ECC are implemented in hardware, on-chip memories, such as low cache levels or load-store queues, may not be protected, cf. [DS16].

2.1 DMR-based error detection

Error detection schemes work by maintaining redundant information that is used to check the integrity of data. This is most evident in error detection schemes based on DMR, where two copies are kept of each data word. If the two copies disagree, an error must have occurred. By comparing the two copies, all single bit flips can be detected. Multiple bit flips can also be detected, provided they do not affect the two copies in identical ways. In particular, multiple bit flips can always be detected if they occur in only one of the copies.

Note that applying error detection by DMR to multi-threaded applications can be problematic. If all memory accesses are duplicated non-selectively, care must be taken to avoid race conditions when different threads access redundant copies of data.

2.2 Error detection by encoding

An alternative approach to error detection is based on encoding data. If the set of valid code words is a small subset of all possible data words, a fault is likely to produce a data word that is not a valid code word. Hence, errors can be detected by checking whether data words are also valid code words. Parity checking is an example of this: in valid code words, the parity bit equals the parity of the code word. This enables the detection of single bit flips. ECC memory typically uses more sophisticated codes, which can also correct errors.

When data is encoded, additional bits are required to represent code words. Although these bits contain redundant information, no data is duplicated explicitly. Therefore, encoding-based error detection schemes can immediately be applied to multi-threaded applications.

A simple, yet effective, encoding-based error detection scheme for integer values can be defined by decreeing that the valid code words are precisely the multiples of a fixed integer constant A. This is known as *AN encoding* [Br60, Fo89]. Variants of AN encoding are popularly used in software-implemented error detection [CRA06, FSS09, Sc10, KF15, Ri15, Ka16]. While AN encoding has the advantage that its capability to detect complex error patterns can be adjusted flexibly by varying the encoding constant A, not all values of A are equally well-suited to error detection [Ho14].

2.3 Memory error detection by AN encoding

We now describe a scheme for detecting errors in memory. The scheme is based on AN encoding, and the key idea is that only valid code words are kept in memory. To achieve this, an integer value m must be *encoded* before being stored:

$$m_{encoded} = m \cdot A. \tag{1}$$

Consequently, whenever a value $m_{encoded}$ is loaded from memory, it must be *decoded* before further processing takes place:

$$m = m_{encoded}/A. \tag{2}$$

Errors can be detected by evaluating the following boolean expression for a value n that has been loaded from memory:

$$n \bmod A = 0. \tag{3}$$

In the absence of errors, the value n is a valid code word, and expression (3) evaluates to `TRUE`. Hence, if expression (3) evaluates to `FALSE`, an error must have occurred.

The presented AN encoding scheme has been implemented by instrumenting load and store instructions in the LLVM IR of programs. Every store instruction is preceded by a multiplication with the constant A, cf. (1). Following every load instruction there is a modulo operation for error checking, cf. (3), and a division for decoding, cf. (2). Figure 1 proves that this approach to memory error detection has its limitations since the compiler backend inserts additional load instructions when lowering the IR to machine code. Errors in memory that affect these load instructions cannot be detected at the IR level.

3 The Extended Compiler Backend

To overcome the limitations of IR-based error detection, the additional memory accesses that are inserted by the compiler backend must be protected against faults. Backends for the C programming language insert memory accesses for the following purposes: to handle register spills (*spill*); to save and restore callee-saved registers (*csr*), the frame pointer (*fptr*), and the return address (*return*); to pass function arguments (*arg*); to access jump tables (*jt*). Since all of these memory accesses, apart from jump table accesses, operate

on the local program stack, duplicating these accesses causes no issues for multi-threaded applications that use shared memory. Since jump table accesses are read-only, their duplication is safe too. We have extended the LLVM backend [LA04] for the *x86* architecture to implement DMR-based error detection for backend-inserted memory accesses.

3.1 Register spills

The *x86* machine code in Listings 1 and 2 illustrates how DMR-based error detection works for register spills. Originally, cf. Listing 1, the register `eax` is spilled to a stack slot at offset –0x30 from the frame pointer (in register `ebp`). The extended backend allocates a second stack slot at offset –0x34, cf. Listing 2. When the register `eax` is restored, the values at the two stack slots are compared. If disagreement is found, control is transferred to an error handler. For the purpose of the present work, error handling consists of exiting the program with a special exit code that indicates that an error has been detected.

List. 1: Register spill and restore.

```
mov   eax,-0x30(ebp)
  ...
mov   -0x30(ebp),eax
add   eax,esi
```

List. 2: Duplicated spill and error checking.

```
mov   eax,-0x34(ebp)
mov   eax,-0x30(ebp)
  ...
mov   -0x30(ebp),eax
cmp   -0x34(ebp),eax
jne   <error_handler>
add   eax,esi
```

3.2 Other stack accesses

The memory accesses *csr*, *fptr*, *return*, and *arg* are analogous to register spills in that they also save values to the program stack and later restore these values to registers. DMR-based error detection is added by duplicating the values on the stack, completely analogously to Listing 2. Full implementation details of DMR-based error detection for these memory accesses can be found in the accompanying technical report [RC16]. Here we only discuss the subtleties of the *return* and *arg* accesses.

On the *x86* architecture, the return address is always passed on the stack. Thus, given the possibility of memory faults, it can never be assumed that the return address is correct. To obtain a copy of the return address that is guaranteed to be correct, the calling convention has been modified so that the return address is passed in a register. Note that on architectures with a designated return register, e.g. ARM or MIPS, protecting the return address against memory faults does not require that the calling convention be modified.

To detect errors in function arguments that are passed on the stack, the calling convention has been modified so that a duplicated copy of the argument sequence is put on the stack immediately above the original sequence, as in Figure 2. When a function argument is loaded into a register, error checking is performed by comparing its value with the corresponding value in the duplicated argument sequence.

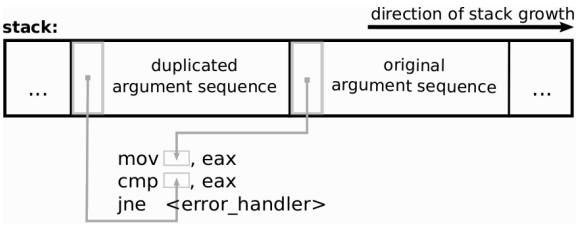


Fig. 2: Original and duplicated function arguments on the stack.

The obligation to implement our modified calling convention rests entirely with the caller. This means that, if a callee chooses not to perform error detection on the return address or on its stack arguments, this does not break function calls. In particular, library functions can still be called fully transparently from within protected functions.

3.3 Jump tables

Jump tables are arrays of addresses of basic blocks, and they reside in the program code segment. To protect jump tables against errors, the extended backend duplicates each jump table in the code segment. Before transferring control to an address that is stored in a jump table, error checking is performed by comparing the address with the corresponding entry in the duplicated jump table.

4 Evaluation

Since faults occur rarely in individual devices, one must actively inject faults into systems or programs to evaluate the effectiveness of fault tolerance schemes. In this work, the test programs from Table 1 are used for this purpose. Some of the test programs (C, E, K) appear in the MiBench suite [Gu01], and similar programs are often used to evaluate fault tolerance schemes [Re99, OSM02, KF15, Ri15, DS16]. The programs represent typical algorithmic tasks, such as sorting, tree and graph traversal, manipulation of bit patterns, and linear algebra. In test program L, a switch statement selects one of the many arguments of the enclosing function; this test has been included here since it is the only one that passes function arguments on the stack for the 64-bit calling convention on *x86*.

Binaries have been generated from the test programs on the *i386* architecture (the 32-bit version of *x86*) and on *x86_64* (the 64-bit version of *x86*). Figure 3 depicts the work flow for this. Program IR is generated by the Clang compiler frontend, which is part of the LLVM infrastructure. Binaries are evaluated on both *i386* and *x86_64* since these architectures have different numbers of general purpose registers: while *x86_64* has sixteen, *i386* has only eight. As a consequence, there will be more backend-inserted memory accesses in *i386* binaries. All binaries have been generated at optimization level `-O3`, and the constant $A = 58659$ has been used for AN encoding, cf. [Ho14].

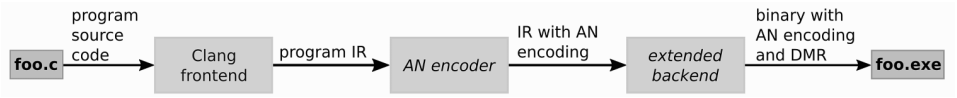


Fig. 3: Generation of binaries with memory error detection measures.

	description
A	array reduction
B	bubblesort
C	cyclic redundancy checker (CRC-32)
D	DES encryption algorithm
E	Dijkstra's algorithm (shortest path)
	arithmetic expression interpreter
F	recursive expression tree evaluation
G	token lexer for arithmetic expressions
H	arithmetic expression parser
I	matrix multiplication
J	array copy
K	quicksort
L	switch

Tab. 1: Suite of test programs.

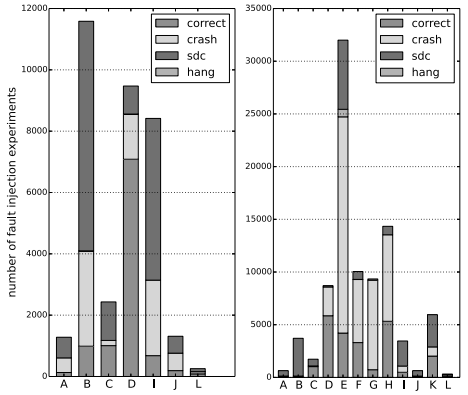
It should be noted that AN encoding produces incorrect programs if the bit width of the constant A is so large that not all encoded values fit into the machine data word. This problem occus on the *i386* architecture when high addresses, e.g. addresses in the stack area, are encoded: multiplying high addresses with the constant A results in a value that cannot be represented by 32 bits. For this reason, AN encoding of the test programs E, F, G, H, K fails on the *i386* architecture. On *x86_64* this is not a problem since pointers are only 48 bits wide and the chosen constant $A = 58659$ is represented by 16 bits.

In evaluating error detection schemes, it is common practice to inject single bit flips, e.g. [YGS09, Fe10, DS16]. To study how programs respond to memory errors, we inject single bit flips into the data words resulting from load operations. This is facilitated by the Intel Pin tool [Lu05] for dynamic binary instrumentation: during the execution of a binary, a single load operation is instrumented with an xor-operation that flips one of the bits in the result of the load. We refer to the execution of a binary with a flipped bit as a *fault injection experiment*. The outcome of a fault injection experiment is determined by the program's response to the injected bit flip. The following responses can occur:

1. *correct*: The program terminates normally and produces correct output.
2. *hang*: The program runs for longer than $10x$ its normal execution time, and is therefore deemed to hang. In practice, especially in safety-critical embedded applications, a hardware watchdog may terminate and restart long-running programs.
3. *crash*: The program terminates abnormally, e.g. due to a segmentation fault.
4. *sdc*: Silent data corruption occurs when the program terminates normally but produces incorrect output.

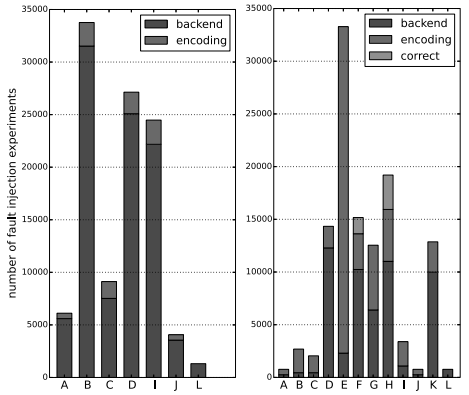
5. *encoding*: The fault is detected by AN encoding.
6. *backend*: The fault is detected by the extended backend's DMR measures.

The fault injection experiments conducted in this work cover all possible patterns in which single bit flips in memory can affect the binaries generated from the test programs. In the following, we therefore report absolute numbers of program responses.



a: *i386* binaries. b: *x86_64* binaries.

Fig. 4: No error detection.



a: *i386* binaries. b: *x86_64* binaries.

Fig. 5: AN encoding and DMR in the backend.

4.1 No error detection

Figure 4 summarizes the fault injection experiments for the *plain* binaries generated from the programs in Table 1, i.e. the binaries without any form of error detection. Only the test programs for which AN encoding produces correct programs appear in Figure 4a.

While *crash* responses indicate that something has gone wrong, when *sdc* occurs in practice, one has no reason to believe that the computed output is incorrect. Therefore, one is often particularly interested in the proportion of *sdc* [Fe10, KF15, DS16]. For the *i386* binaries the number of *sdc* is generally larger than for *x86_64*. This is to be expected given that *i386* has fewer registers: more data words that are relevant for the program output will, at least temporarily, reside in memory and hence be vulnerable to faults.

Figure 4 shows that there is indeed a need for error detection schemes. While some faults do not affect program behavior, thus leading to *correct* responses, there is always a large fraction of malignant program responses, i.e. *crash*, *hang*, and *sdc*.

4.2 AN encoding with DMR in the compiler backend

When AN encoding at the IR level is combined with the extended compiler backend, all single bit flips in memory are detected, as evidenced by Figure 5. For the binaries F and

H on *x86_64* there are a number of *correct* responses, which occur when faults affect load operations that are part of a call to the `memcpy` library function. Although this function call is present in the IR, it is not protected by our AN encoding scheme since no data is loaded into the program. The response *correct* ensues when faults affect only those portions of the copied data that are subsequently not used and hence not loaded into the program.

While Figure 5a, for the 32-bit binaries, is dominated by *backend* responses, this is not the case for Figure 5b. Since *x86_64* has more general purpose registers than *i386*, there is lower register pressure, causing the backend to insert fewer additional memory accesses to handle *spill*, *csr* etc. Also note that the total numbers of fault injection experiments in Figure 5a are about twice as high as in Figure 4a. This is due to the dominating *backend* responses in Figure 5a and the fact that the extended backend duplicates memory accesses.

Figure 5 proves that our approach to detecting memory errors is effective: no malignant program responses remain. In particular, DMR-based error detection in the extended compiler backend succeeds at removing the vulnerabilities that Figure 1 hints at.

4.3 Runtime overheads

Fault tolerance comes at the price of performance penalties since some form of redundancy is required. The runtimes of the test programs from Table 1 are depicted in Figure 6, where geometric means across all test programs are shown. Runtimes have been normalized to the *plain* binaries, without any error detection measures.

The largest fraction of overhead is due to AN encoding, which is plausible given the high latency of integer multiplication, division, and modulo, cf. (1)–(3). In fact, AN encoding is known to introduce large overheads, cf. [FSS09, Ri15, Ka16]. Given that AN encoding is responsible for detecting only a small fraction of errors in the *i386* binaries, cf. Figure 5a, the overhead that AN encoding introduces may not be justifiable on *i386*.

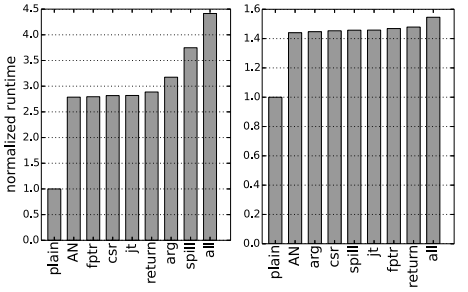
The overhead that the extended backend causes on *i386* is dominated by *spill*, followed by *arg*. This is in agreement with the fact that *i386* has relatively few registers and uses a calling convention by which all arguments are passed on the stack. Neither of these observations apply to the *x86_64* architecture, and hence the overhead introduced by the extended backend is considerably lower.

The runtime overhead introduced by the extended backend has also been evaluated on a subset of the SPEC CINT2006 suite. The subset consists of those C benchmarks that are unaffected by our modified calling convention, which are: `400.perlbench`, `401.bzip2`, `429.mcf`, `445.gobmk`, `458.sjeng`, `462.libquantum`. Geometric means across these benchmarks are shown in Figure 7, where runtimes have again been normalized to the *plain* binaries, to which the backend has not applied any DMR-based error detection measures. Note that the overhead introduced by duplicating function arguments is lower in Figure 7a than in Figure 6a. An explanation for this is that functions in the SPEC benchmarks have larger bodies, and hence longer execution times, than in the test programs from Table 1. Therefore, the overhead introduced by duplicated function arguments car-

ries less weight. When *all* DMR measures are applied by the backend, the resulting mean overheads are 1.50*x* on *i386* and 1.13*x* on *x86_64*.

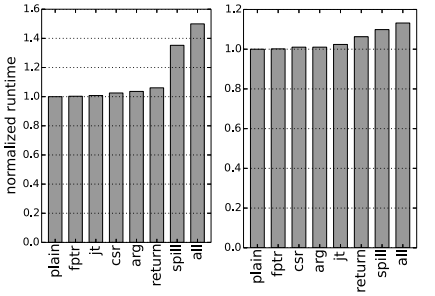
From Figures 6 and 7 it is clear that the handling of register spills is the dominant source of overhead among the DMR-based measures in the compiler backend. Comparing the results for the *i386* and *x86_64* architectures, it can be concluded, perhaps unsurprisingly, that memory error detection is more efficient on architectures with many registers. It should also be noted that the overhead of handling return addresses can be reduced, if not entirely avoided, on architectures that pass the return address in a register, e.g. ARM or MIPS.

All runtime measurements were conducted on an Intel Core i7-4790 CPU (3.6GHz), with 32GB of main memory. The operating system is Ubuntu 16.04.1 LTS, with a 4.4.0 kernel.



a: *i386* binaries. b: *x86_64* binaries.

Fig. 6: Mean overheads for test programs.



a: *i386* binaries. b: *x86_64* binaries.

Fig. 7: Mean overheads for SPEC.

5 Related work

Fault tolerance schemes that operate on program source code appeared early, and this approach is still pursued [Re99, BSS12, BSS13, KF15, Ka16]. Low rates of silent data corruption can be achieved despite the fact that the compiler backend may introduce new vulnerabilities after these schemes have been applied. However, a considerable proportion of faults still lead to program crashes, e.g. [KF15].

With the advent of super-scalar processors it became viable to implement DMR-based error detection schemes by duplicating machine instructions [OSM02]. Subsequently proposed fault tolerance schemes were also implemented by modifying compiler backends, e.g. [Re05, YGS09, MPC14, DS16]. Unlike in the present article, these schemes assume that memory is protected by hardware measures, e.g. ECC, and hence memory operations are not accompanied by error detection. The only exception to this is the nZDC scheme [DS16], where memory accesses are duplicated since the processor's load-store queue is assumed to be vulnerable to faults. Since the nZDC scheme duplicates all load operations, and not just those that access local memory, it is limited in handling multi-threaded applications correctly, as already noted in [DS16].

The popularity of the LLVM compiler framework and IR [LA04] has led to many IR-based fault tolerance schemes [FSS09, Sc10, Fe10, Zh10, Ri15, CNV16]. The DMR-based schemes [Fe10, Zh10, CNV16] assume, once again, that memory is protected against faults by hardware measures. The encoding-based schemes [FSS09, Sc10, Ri15] do not make this assumption, but they suffer from the observed shortcoming that the memory accesses that are introduced by the compiler backend are left unprotected.

That return addresses and frame pointers need protection was already observed in the context of protecting an operating system against hardware faults [BSS13]. The fault tolerance scheme in [BSS13] was implemented based on *aspects* [SGSP02]. Conceptually, aspects operate on program source code, but their implementation requires interaction with the compiler. Thus, the implementation of aspects may introduce new vulnerabilities.

AN encoding was originally introduced in [Br60] and studied in detail, among other *arithmetic error codes*, by [Ga66, Av71]. Protecting processors by AN encoding was suggested in [Fo89], where the ANB and ANBD schemes were also introduced. IR-based implementations of AN encoding appeared in [FSS09, Ri15]. As in the present article, other fault tolerance schemes also combine encoding with DMR [OMM02, CRA06, KF15]. Here we applied DMR very selectively, only to local memory accesses inserted by the compiler backend, which has the advantage that duplication is safe in multi-threaded programs.

6 Summary and Discussion

Fault tolerance schemes that are applied to programs at the level of intermediate representation (IR) cannot address vulnerabilities resulting from later stages of the compilation process. Specifically, the compiler backend introduces additional, unprotected memory accesses to implement, e.g., register spills. In this article we have presented an extended backend that adds error detection by dual modular redundancy (DMR) to the memory accesses it inserts. It has been shown that this, combined with an IR-based AN encoding scheme, succeeds at detecting all errors resulting from single bit flips in memory.

The extended backend introduces an average runtime overhead of $1.50x$ for binaries from SPEC CINT2006 running on *i386*, and $1.13x$ for the corresponding binaries running on *x86_64*. This is in agreement with the expectation that there is less need to protect against faults in the memory system on machines with more registers, i.e. on *x86_64*. The reported runtime overheads are noticeably lower than for the nZDC scheme, which also duplicates memory accesses [DS16]. This is unsurprising since, in the present work, error detection has been applied to memory accesses more selectively.

Implementing fault tolerance schemes at the IR level enables target-independence and enhances productivity. The latter is particularly important for relaxed fault tolerance schemes, where some amount of vulnerability is accepted in exchange for reduced overhead [Fe10, KF15, Ri15]. In quantifying the vulnerabilities of a relaxed scheme, meaningful results can only be obtained if one is guaranteed that the compilation process following the application of the fault tolerance scheme does not introduce new vulnerabilities. The extended backend we have presented here gives this guarantee.

Due to strict safety and reliability requirements, automotive applications may not be able to rely on relaxed fault tolerance schemes. Schemes based on IR transformations only are also not an option due to the remaining vulnerabilities. The extended compiler backend overcomes this problem and thus facilitates complete memory error detection.

7 Acknowledgments

This work was funded by the German Research Council (DFG) through the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed). The authors acknowledge useful discussions with Sven Karol and Tobias Stumpf.

References

[Av71] Avizienis, A.: Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. IEEE Trans. on Computers, C-20(11):1322–1331, 1971.

[Ba05] Baumann, R.: Soft Errors in Advanced Computer Systems. IEEE Design & Test of Computers, 22(3):258–266, 2005.

[Bl06] Blome, J. A.; Gupta, S.; Feng, S.; Mahlke, S.: Cost-efficient soft error protection for embedded microprocessors. In: Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems. CASES'06, pp. 421–431, 2006.

[Bo05] Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro, 25(6):10–16, 2005.

[Br60] Brown, D. T.: Error Detecting and Correcting Binary Codes for Arithmetic Operations. IRE Trans. Electronic Computers, pp. 333–337, 1960.

[BSS12] Borchert, C.; Schirmeier, H.; Spinczyk, O.: Protecting the Dynamic Dispatch in C++ by Dependability Aspects. In: Proc. 1st Workshop Software-Based Methods for Robust Embedded Systems. SOBRES'12, 2012.

[BSS13] Borchert, C.; Schirmeier, H.; Spinczyk, O.: Return-Address Protection in C/C++ Code by Dependability Aspects. In: Proc. 2nd Workshop Software-Based Methods for Robust Embedded Systems. SOBRES'13, 2013.

[CNV16] Chen, Z.; Nicolau, A.; Veidenbaum, A. V.: SIMD-based Soft Error Detection. In: Proc. ACM Int'l Conf. Computing Frontiers. CF'16, pp. 45–54, 2016.

[CRA06] Chang, J.; Reis, G. A.; August, D. I.: Automatic Instruction-Level Software-Only Recovery. In: Int'l Conf. Dependable Systems and Networks. DSN'06, pp. 83–92, 2006.

[DS16] Didehban, M.; Shrivastava, A.: nZDC: A compiler technique for near Zero Silent Data Corruption. In: Proc. Design Automation Conf. DAC'16, 2016.

[Es11] Esmaeilzadeh, H.; Blem, E.; Amant, R. St.; Sankaralingam, K.; Burger, D.: Dark silicon and the end of multicore scaling. In: Proc. 38th Ann. Int'l Symp. Computer Architecture. ISCA'11, pp. 365–376, 2011.

[Es12] Esmaeilzadeh, H.; Sampson, A.; Ceze, L.; Burger, D.: Architecture Support for Disciplined Approximate Programming. In: Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems. ASPLOS'12, pp. 301–312, 2012.

[Fe10] Feng, S.; Gupta, S.; Ansari, A.; Mahlke, S.: Shoestring: Probabilistic Soft Error Reliability on the Cheap. In: Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems. ASPLOS'10, pp. 385–396, 2010.

[Fo89] Forin, P.: Vital Coded Microprocessor Principles and Applications for Various Transit Systems. In: Control, Computers, Communications in Transportation: Selected Papers from the IFAC/IFIP/IFORS Symposium. pp. 79–84, 1989.

[FSS09] Fetzer, C.; Schiffel, U.; Süßkraut, M.: AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. In: Proc. 28th Int'l Conf. Computer Safety, Reliability, and Security. SAFECOMP'09, pp. 283–296, 2009.

[Ga66] Garner, H. L.: Error Codes for Arithmetic Operations. IEEE Trans. Electronic Computers, EC-15(5):763–770, 1966.

[Gu01] Guthaus, M. R.; Ringenberg, J. S.; Ernst, D.; Austin, T. M.; Mudge, T.; Brown, R. B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: Proc. IEEE Int'l Symp. Workload Characterization. IISWC'01, pp. 3–14, 2001.

[Ho14] Hoffmann, M.; Ulbrich, P.; Dietrich, C.; Schirmeier, H.; Lohmann, D.; Schröder-Preikschat, W.: A Practitioner's Guide To Software-based Soft-Error Mitigation Using AN-Codes. In: Proc. 15th Int'l Symp. High-Assurance Systems Engineering. 2014.

[HSS12] Hwang, A. A.; Stefanovici, I. A.; Schroeder, B.: Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In: Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems. ASPLOS'12, pp. 111–122, 2012.

[Ka16] Karol, S.; Rink, N. A.; Gyapjas, B.; Castrillon, J.: Fault Tolerance with Aspects: A Feasibility Study. In: Proc. 15th Int'l Conf. Modularity. 2016.

[KF15] Kuvaiskii, D.; Fetzer, C.: Δ-encoding: Practical Encoded Processing. In: Proc. 45th Ann. Int'l Conf. Dependable Systems and Networks. DSN'15, 2015.

[LA04] Lattner, C.; Adve, V.: LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. Int'l Symp. Code Generation and Optimization. CGO'04, p. 75, 2004.

[Li11] Liu, S.; Pattabiraman, K.; Moscibroda, T.; Zorn, B. G.: Flikker: saving DRAM refresh-power through critical data partitioning. In: Proc. 16th Int'l Conf. on Architectural Support for Programming Languages and Operating systems. ASPLOS'11, pp. 213–224, 2011.

[Lu05] Luk, C.-K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; S. Wallace, Steven; Reddi, V. J.; Hazelwood, K.: PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proc. Conf. Programming Language Design and Implementation. PLDI'05, pp. 190–200, 2005.

[MPC14] Mitropoulou, K.; Porpodas, V.; Cintra, M.: DRIFT: Decoupled CompileR-Based Instruction-Level Fault-Tolerance. In: Proc. 26th Int'l Workshop Languages and Compilers for Parallel Computing. LCPC'13, pp. 217–233, 2014.

[NDO11] Nightingale, E. B.; Douceur, J. R.; Orgovan, V.: Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In: Proc. 6th Conf. on Computer Systems. EuroSys'11, pp. 343–356, 2011.

[OMM02] Oh, N.; Mitra, S.; McCluskey, E. J.: ED4I: Error Detection by Diverse Data and Duplicated Instructions. IEEE Trans. Computers, 51(2):180–199, 2002.

[OSM02] Oh, N.; Shirvani, P. P.; McCluskey, E. J.: Error Detection by Duplicated Instructions in Super-Scalar Processors. IEEE Trans. Reliability, 51(1):63–75, 2002.

[Pa08] Panaroni, P.; Sartori, G.; Fabbrini, F.; Fusani, M.; Lami, G.: Safety in Automotive Software: An Overview of Current Practices. In: Proc. 32nd Ann. IEEE Int'l Computer Software and Applications Conf. COMPSAC'08, pp. 1053–1058, 2008.

[RC16] Rink, N. A.; Castrillon, J.: Comprehensive Backend Support for Local Memory Fault Tolerance. Technical Report TUD-FI-16-04, Technische Universität Dresden, 2016.

[Re99] Rebaudengo, M.; Reorda, M. S.; Torchiano, M.; Violante, M.: Soft-error Detection through Software Fault-Tolerance techniques. In: Int'l Symp. Defect and Fault Tolerance in VLSI Systems. DFT'99, pp. 210–218, 1999.

[Re05] Reis, G. A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D. I.: SWIFT: Software Implemented Fault Tolerance. In: Int'l Symp. Code Generation and Optimization. CGO '05, pp. 243–254, 2005.

[Ri15] Rink, N. A.; Kuvaiskii, D.; Castrillon, J.; Fetzer, C.: Compiling for Resilience: The Performance Gap. In: Proc. Mini-Symp. Energy and Resilience in Parallel Programming. ERPP'15, 2015.

[Sc10] Schiffel, U.; Schmitt, A.; Süßkraut, M.; Fetzer, C.: ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In: Proc. 29th Int'l Conf. Computer Safety, Reliability, and Security. SAFECOMP'10, pp. 169–182, 2010.

[SGSP02] Spinczyk, O.; Gal, A.; Schröder-Preischkat, W.: AspectC++: An aspect-oriented extension to the C++ programming language. In: Proc. 40th Int'l Conf. Tools Pacific: Objects for internet, mobile and embedded applications. CRPIT'02, pp. 53–60, 2002.

[Sh02] Shivakumar, P.; Kistler, M.; Keckler, S. W.; Burger, D.; Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proc. Int'l Conf. Dependable Systems and Networks. DSN'02, pp. 389–398, 2002.

[Sh14] Shafique, M.; Garg, S.; Henkel, J.; Marculescu, D.: The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In: Proc. 51st Ann. Design Automation Conf. DAC'14, pp. 1–6, 2014.

[SPW09] Schroeder, B.; Pinheiro, E.; Weber, W.-D.: DRAM Errors in the Wild: A Large-scale Field Study. In: Proc. 11th Int'l joint Conf. Measurement and Modeling of Computer Systems. SIGMETRICS'09, pp. 193–204, 2009.

[Ta12] Taylor, M. B.: Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In: Proce. 49th Ann. Design Automation Conf. DAC'12, pp. 1131–1136, 2012.

[We15] Weis, C.; Jung, M.; Ehses, P.; Santos, C.; Vivet, P.; Goossens, S.; Koedam, M.; Wehn, N.: Retention Time Measurements and Modelling of Bit Error Rates of WIDE I/O DRAM in MPSoCs. In: Proc. Design, Automation & Test in Europe Conf. & Exhibition. DATE'15, pp. 495–500, 2015.

[YGS09] Yu, J.; Garzarán, M. J.; Snir, M.: ESoftCheck: Removal of Non-vital Checks for Fault Tolerance. In: Proc. 7th Ann. Int'l Symp. Code Generation and Optimization. CGO'09, pp. 35–46, 2009.

[Zh10] Zhang, Y.; Lee, J. W.; Johnson, N. P.; August, D. I.: DAFT: Decoupled Acyclic Fault Tolerance. In: Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques. PACT'10, pp. 87–98, 2010.